Randomizing TCP payload size for TCP fairness in data center networks

Citation for published version:
Lee, S, Lee, D, Lee, M, Jung, H & Lee, B 2017, 'Randomizing TCP payload size for TCP fairness in data center networks', *Computer Networks*, vol. 129, pp. 79-92. https://doi.org/10.1016/j.comnet.2017.09.007

Digital Object Identifier (DOI):
10.1016/j.comnet.2017.09.007

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Peer reviewed version

Published In:
Computer Networks

Randomizing TCP Payload Size for TCP Fairness in Data Center Networks☆

Soojeon Lee[a,b], Dongman Lee[a,*], Myungjin Lee[c], Hyungsoo Jung[d], Byoung-Sun Lee[b]

[a]*School of Computing, KAIST, Daejeon, 34141, South Korea*
[b]*Aerospace System Research Section, ETRI, Daejeon, 34129, South Korea*
[c]*School of Informatics, University of Edinburgh, Edinburgh EH8 9AB, United Kingdom*
[d]*Division of Computer Science and Engineering, Hanyang University, 04763, South Korea*

Abstract

As many-to-one traffic patterns prevail in data center networks, TCP flows often suffer from severe unfairness in sharing bottleneck bandwidth, which is known as the TCP outcast problem. The cause of the TCP outcast problem is the bursty packet losses by a drop-tail queue that triggers TCP timeouts and leads to decreasing the congestion window. This paper proposes TCPRand, a transport layer solution to TCP outcast. The main idea of TCPRand is the randomization of TCP payload size, which breaks synchronized packet arrivals between flows from different input ports. Based on the current congestion window size and the CUBIC's congestion window growth function, TCPRand adaptively determines the proper level of randomness. With extensive ns-3 simulations and experiments, we show that TCPRand guarantees the superior enhancement of TCP fairness by reducing the TCP timeout period noticeably even in an environment where serious TCP outcast happens. TCPRand also minimizes the total goodput loss since its adaptive mechanism avoids unnecessary payload size randomization. Compared with DCTCP, TCPRand performs fairly well and only requires modification at the transport layer of the sender which makes its deployment relatively easier.

Keywords: Data center networks, TCP outcast, fairness.

1. Introduction

Data center applications such as MapReduce and network file systems create a many-to-one traffic pattern that is bursty and barrier-synchronized. In such a traffic pattern, multiple TCP flows arrive at different input ports of a bottleneck switch and compete for the same outgoing queue. This makes those data center applications suffer from serious goodput decrease and bad flow completion time performance due to frequent TCP timeouts triggered by multiple packet losses. Such a phenomenon can lead to the well-known TCP incast [1] and outcast problems [2].

The TCP incast problem typically occurs when TCP senders and a receiver are colocated on the same rack (or under one aggregate switch) and all flows go out through the same shallow-buffered switch [1, 3, 4, 5]. TCP timeouts happen randomly among flows competing for the outgoing port at a bottleneck switch. In contrast, the TCP outcast problem arises when the locations of the senders are dispersed across different racks. Specifically, the bottleneck switch penalizes particular flows more often than others by consecutively dropping more packets from those flows. Thus, the outcast problem severely hurts TCP fairness—a crucial metric, especially for barrier-synchronized workloads in data center networks. In such workloads, speeding up the slowest flow in a barrier is a key to enhance

☆Portions of this manuscript appeared in IEEE Infocom 2015. This work was supported in part by the Space Core Technology Development Program of NRF [NRF-2014M1A3A3A03034729, Development of core S/W standard platform for GEO satellite ground control system], ICT R&D program of MSIP/IITP [B0101-14-0334, Development of IoT-based Trustworthy and Smart Home Community Framework], and a grant from the British Council.

*Corresponding author

Email addresses: soojeonlee@{kaist.ac.kr, etri.re.kr} (Soojeon Lee), dlee@cs.kaist.ac.kr (Dongman Lee), myungjin.lee@ed.ac.uk (Myungjin Lee), hyungsoo.jung@hanyang.ac.kr (Hyungsoo Jung), lbs@etri.re.kr (Byoung-Sun Lee)

the performance since a barrier ends only after every flow (including the slowest one) in the barrier finishes. TCP outcast is attributed to burst arrivals of packets competing for the same output port and a severe imbalance in the number of incoming flows per each input port at the bottleneck switch. Because inter-rack sender placement begins to be taken into account in order to improve fault tolerance in data centers [6], it is of utmost importance for data center administrators to have a viable solution to the outcast problem.

Several solutions can be applied for the TCP outcast problem. The link layer solutions require a modication to the current switching architecture [7] or are not widely supported in todays switches [8]. Equal-length routing [2], one of network layer solutions, only works in non-oversubscribed networks. The cross-layer solution [9] levereages the Explicit Congestion Notication (ECN) capability, which is becoming popular at the world largest data centers. However, there still exist many data centers where all or majority of switches do not have ECN capability due to cost reason; in fact the largest data centers in South Korea are an exemplar of this reason. To help such data centers overcome the TCP outcast problem efficiently, a transport layer solution can be viable since it neither relies on any specific link layer supports nor assumes any particular network topology. However, existing rate-based transport layer approaches are not applicable to TCP outcast in data centers because they require the precise control of inter packet spacing time [10, 11] which operating systems hardly guarantee and are inappropriate [11] for a multi-hop environment.

The contribution of this paper is a transport layer solution called *TCPRand* to TCP outcast. TCPRand randomizes the TCP payload size to break the bursty arrival times of back-to-back packets. By doing so, it prevents the outcast flow suffering from consecutive packet losses and consequently reduces TCP timeouts. At the sender side, the proposed solution makes the TCP payload size uniformly distributed between [*rMin*, MSS]. However, it may increase the packet header overhead due to the smaller payload size and curtail the total goodput. To achieve high fairness without loss of total goodput, it calculates *rMin* by adapting to the changes of congestion window (*cwnd*). It is based on the observation that for many-to-one applications (e.g., especially with a barrier synchronization property [12]) as *cwnd* of a flow is growing, the network is more congested and the port blackout happens more frequently. Hence, if *cwnd* of a flow increases, the scheme decreases *rMin* for the flow.

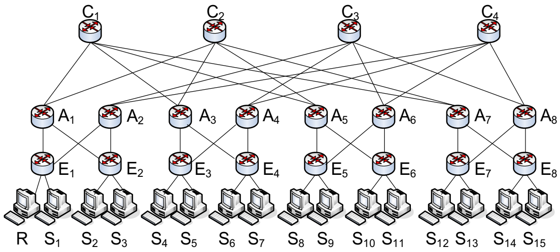


Figure 1: Fat-tree topology composed of switches (C_n: Core, A_n: Aggregation, and E_n: Edge) and end-nodes (R: Receiver and S_n: Sender).

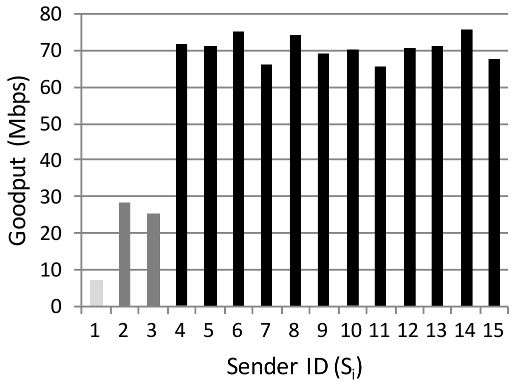


Figure 2: The goodput of each flow sent from the 15 senders described in Figure 1. The flow sent from S_1 is the most outcast one and the flows from S_2-S_3 are the second outcast ones.

We use ns-3 [13] to evaluate TCPRand with a realistic topology (i.e., fat-tree [14]) and workloads of data center networks, and show that TCPRand substantially improves TCP fairness and rarely sacrifices flow completion times of flows, especially those of small flows. In addition, we implement TCPRand by modifying the sender side execution path of TCP protocol stack in the Linux kernel and perform extensive experiments in our testbed. We demonstrate that TCPRand reduces consecutive packet drops and TCP timeouts significantly, and as a result, it improves TCP fairness substantially with a small loss of the overall goodput and negligible additional retransmission overheads. We also compare TCPRand with DCTCP (i.e., the most popular solution with support of switch) and shows that TCPRand increases fairness as close to 98% of DCTCP.

The remainder of this paper is organized as follows. In Section 2, we briefly introduce the TCP outcast problem. Next, we explain the effect of payload size randomization and why it is a key technique to the outcast problem in Section 3. Section 4 provides the details of TCPRand. We outline our evaluation setup in Section 5 and evaluation results are presented in Sections 6 and 7. Related works are discussed in Section 9 before we conclude in Section 10.

2. The TCP Outcast Problem

2.1. Overview

The TCP outcast problem is observed easily in data center networks, where routers or switches are usually connected through a multi-rooted and hierarchical topology such as fat-tree [14] and senders and receivers are leaves of a topology. For instance in Figure 1, there are 15 senders (i.e., S_1-S_{15}) and a receiver (i.e., R). As many-to-one delivery applications emerge in such an environment, multiple flows arrive at different input ports of a receiver's ingress switch (i.e., E_1) and compete to enter the same outgoing queue. If many flows and a few flows are arriving at two input ports ($A_1 \to E_1$ and $A_2 \to E_1$) and destined to the same output port ($E_1 \to R$), the latter (i.e., the outcast flows) loses the goodput tremendously because TCP timeout is triggered more easily to them. It is called the TCP outcast problem [2], leading to a serious TCP unfairness among flows. For instance, as shown in Figure 2, it even results in much higher goodput decrease in the flows with a short RTT (i.e., from S_1) than in those with a long RTT (i.e., from S_4-S_{15}) in a fat-tree topology.

2.2. Port Blackout

With excessive traffic flows, drop-tail queueing may drop a series of consecutive packets at each input port, and this is called port blackout [2]. We refer to [2] for more details on the phenomenon and here briefly explain it with an example depicted in Figure 3. The figure illustrates how the port blackout occurs at a bottleneck switch where a drop-tail queue management policy is applied and there exist two input ports (i.e., X and Y) and one output port (i.e., Z). We consider a case where the packets of TCP-based bulk data transfer applications arrive at the switch through ports X and Y and leave it via port Z.

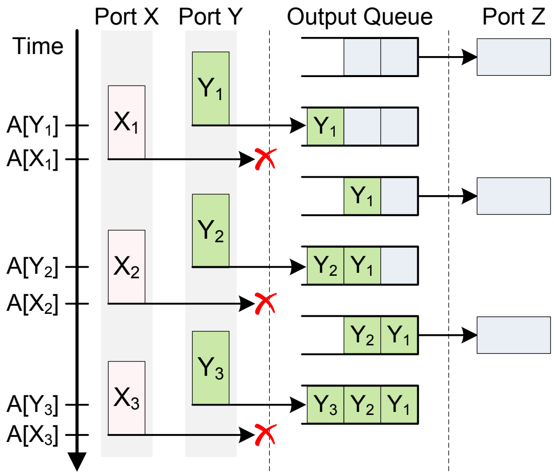


Figure 3: Port blackout at a switch with fixed-size payloads. Synchronized packet arrivals make packets arriving at a particular port (port X in the figure) get discarded with high probability as the output queue is almost always full when they arrive. $A[p]$ denotes an arrival time of packet p at the output queue.

In this setup, packets are almost of the same size (i.e., the size of TCP/IP headers + MSS). Traffic is bursty and the inter-frame gap between packets is constant (e.g., 0.096μs for a gigabit Ethernet according to the IEEE 802.3 specification [15]). This condition can create a situation where packets from port Y are always stored in the output queue while packets from port X are always discarded. This occurs because packet arrivals are almost synchronized and packets from port Y always arrive slightly ahead of competing packets from port X whenever one MSS worth of buffer space becomes available at the output queue. For instance, as shown in Figure 3, the arrival time of packet Y_1 (denoted as $A[Y_1]$) is ahead of that of packet X_1 (i.e., $A[X_1]$), $A[Y_2] < A[X_2]$, and so forth. Even though a series of packet drops happen fairly on ports X and Y by turns, they damage more seriously to the throughput of the incoming stream from port X if the stream consists of less number of TCP flows. As a consequence, the TCP flows from port X are outcast; they experience more frequent TCP timeouts and lose the goodput more substantially than those from port Y. This is the essence of the TCP outcast problem [2] that has negative impacts on the TCP fairness among competing flows from different input ports. It even leads to much higher goodput decrease in flows with a short RTT than in those with a long RTT in a fat-tree topology.

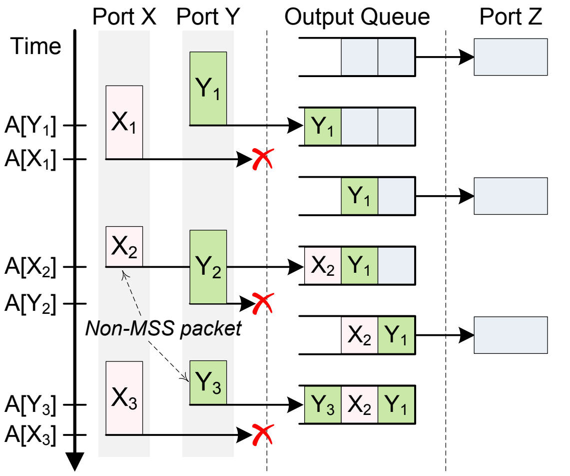


Figure 4: An illustration of packet payload size randomization. The packet payload size randomization technique creates packets with smaller payload size than MSS, and breaks the synchronized arrivals of packets, thereby alleviating the impact of the port blackout phenomenon. In the figure, packet X_2 arrives earlier than Y_2 and finds the output queue is not full. Hence, X_2 is successfully inserted in the queue, as opposed to the result in Figure 3. $A[p]$ denotes an arrival time of packet p at the output queue.

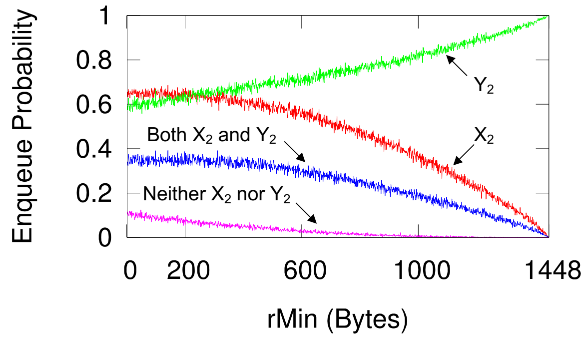


Figure 5: Enqueue probability of X_2 and Y_2 at congestion.

3. Payload Size Randomization

Addressing the TCP outcast problem requires to reduce the consecutive packet losses at each input port, thereby preventing the port blackout, the main cause of TCP outcast. In this section, we introduce a payload size randomization idea which breaks the bursty and synchronized back-to-back packet arrivals and as a consequence mitigates the port blackout phenomenon. Then, through an experiment, we quantitatively show that the randomization method substantially mitigates the degree of the port blackout.

3.1. Avoiding Concurrent Packet Arrivals

The port blackout problem can be ameliorated by reducing concurrent packet arrivals at two input ports. At the transport layer, this can be achieved by the rate-based approach but it is less practical (see Section 9). Our approach to the problem is rather to randomize the size of each TCP payload. The intuition behind this is, randomizing the size of TCP payload can induce randomness in the arrival times of packets and it finally breaks the synchronized arrival times of back-to-back packets at each input port. This can reduce the chance of having port blackout, and the initial randomness can be preserved all the way down to the receiver in multi-hop environments.

Let us consider an example illustrated in Figure 4. In the example, X_1 is dropped because Y_1 arrives slightly before X_1 (i.e., $A[Y_1] < A[X_1]$) when one MSS worth of buffer space is left at the output queue. Next, the payload size randomization technqiue creates a case where the payload size of X_2 is smaller than that of Y_2. This results in the earlier arrival of X_2 than Y_2 (i.e., $A[X_2] < A[Y_2]$). Hence, X_2 is inserted to the output queue whereas Y_2 is dropped (c.f., the opposite phenomenon in Figure 3 due to the port-blackout). The technique again affects the dynamics of the arrival times of X_3 and Y_3 and this time lets Y_3 inserted to the queue and X_3 discarded. Because the payload randomization technique effectively breaks the synchronized arrivals of packets, packet drops occur rather alternately across input ports; thus the frequency of the port blackout phenomenon decreases.

3.2. Understanding the Effect of Payload Size Randomization

To take a closer look at the port blackout phenomenon, we investigate how much a series of packet drops from each input port can be alleviated with the payload size randomization at a switch under congestion. Let Q ($0 \leq Q \leq Q_{max}$) be the output queue length. A packet drop occurs at a drop-tail queue if a packet arrives when $Q = Q_{max}$. To quantitatively measure the effect of the payload size randomization, we focus on the enqueue probability of two packets X_2 and Y_2 after Y_1 is enqueued and X_1 is dropped (see Figure 4). More formally, the probability of packet pkt to be enqueued at $A[pkt]$, is acquired by:

$$P_q(pkt) = 1 - P(Q = Q_{max} \ at \ A[pkt]) \quad (1)$$

Based on the notion of Eq. 1, we experimentally measure the enqueue probabilities of *i)* X_2, *ii)* Y_2, *iii)* both X_2 and Y_2, and *iv)* neither X_2 nor Y_2 while randomizing payload sizes. To do so, we write an offline test code generating two virtual back-to-back flows (from X and Y). We randomly select a payload size of each packet within the range of [*rMin*, MSS]. We vary the degree of randomness by changing *rMin* from 1B to 1,448B at the interval of 1B. We construct a simple experimental setup as follows: First, nodes are connected with 1Gbps links. Second, there are two input ports X and Y, and one output port Z. Third, back-to-back packets arrive continuously at each input port and the inter-frame gap is 0.096μs (i.e., 8B in a gigabit Ethernet). Last, Y_1 is enqueued to the output queue while X_1 is dropped.

By tracing all the packet arrivals and departures since $A[Y_1]$, we measure $P_q(X_2)$ and $P_q(Y_2)$. We conduct this test 1,000 times per each *rMin*. Figure 5 shows the four types of probabilities of interest. If the regular TCP (i.e., the payload size is not randomized at all and *rMin* = 1,448B) is used, X_2 never be enqueued. Of course, this simple experimental result may not hold in real network environments since the packet arrival time can be distorted due to some random factors (e.g., variations in sending patterns or other unpredictable random behaviors) [2, 16] and TCP does not generate endless bursty traffic unlike we did for the test. However, Figure 5 clearly indicates why the port blackout is hard to be prevented with the regular TCP at a drop-tail queueing switch.

As *rMin* decreases (i.e., from the right of the x-axis to the left in Figure 5, $P_q(X_2)$ increases and $P_q(Y_2)$ decreases. $P_q(X_2)$ and $P_q(Y_2)$ approach to 0.63 and 0.58, respectively when *rMin* = 1B. One interesting observation is that the enqueue probability of both X_2 and Y_2 also increases by decreasing *rMin*. However, the payload size randomization can make both X_2 and Y_2 dropped (e.g., with the probability of 0.11 when *rMin* = 1B). Nevertheless, the advantages far outweigh this disadvantage since the probability of consecutive packet drops reduces significantly by the randomization mechanism.

Another implication from the above result is that it is unnecessary to reduce *rMin* overmuch. There are two reasons. First, the enqueue probability of X_2 grows up more slowly as *rMin* approaches to 1B. Second, the lower *rMin*, the larger the header overhead. It results in bandwidth waste.

4. Proposed Scheme: TCPRand

In this section, we focus on the design of our proposed scheme that we call *TCPRand*. Before sending a packet, TCPRand determines its payload size via generating a uniform random number in the range of [*rMin*, MSS]. Since *rMin* is a configurable variable ($1 \leq rMin \leq$ MSS), we can diversify randomly generated payload sizes by selecting one *rMin* value. However, it is unclear what value to set. Moreover, the degree of port blackout can vary depending on several factors such as background traffic, changes in traffic patterns, etc. Due to these reasons, we consider a scheme that can adaptively select *rMin* value and effectively react to changes in such factors. *Our design choice for the adaptation method lies not only in maximizing the fairness, but also in minimizing the loss of total goodput in any circumstances.* We design our adaptation method on top of TCP CUBIC [17], the default congestion control algorithm in Linux.

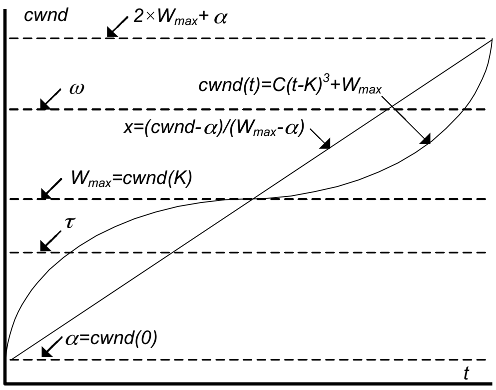
4.1. Modeling Adaptive Selection of rMin in CUBIC

We focus on CUBIC's *cwnd* growth function for designing an adaptive *rMin* selection method as variation in *cwnd* value can be indicative of the probability of packet loss, which is a necessary condition of the port blackout.

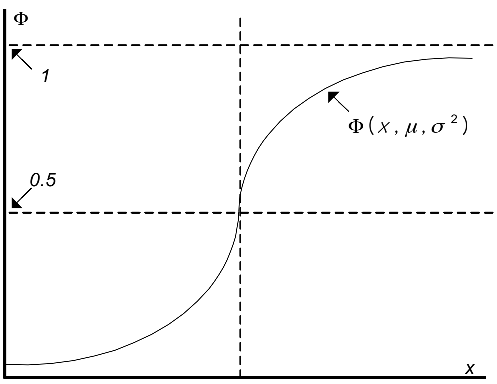
Let us first take a look at how the CUBIC's window growth function (i.e., *cwnd*(t)), depicted in Figure 6(a), works. We classify a CUBIC epoch into 4 stages and present our adaptive *rMin* selection strategy for each stage based on its functional characteristics.

Stage 1) Fast growth of *cwnd* (when *cwnd* < W_{max}): At the initial phase of a CUBIC epoch, the *cwnd* grows very fast. The rationale here is that the fast *cwnd* growth is unlikely to cause a packet drop since the *cwnd* is already reduced by a factor of β just before the start of this epoch. Therefore, as **Strategy 1**, we propose to not reduce *rMin* aggressively.

Stage 2) Slow growth of *cwnd* (when *cwnd* < W_{max}): CUBIC slows down the growth of *cwnd* as approaching to W_{max} since packet losses occurred



(a) CUBIC's *cwnd*(t). In CUBIC, when a packet drop is detected, the *cwnd* decreases by a factor of β (= 717/1024 in Linux kernel). Then, a new CUBIC epoch begins at t=0, and the initial *cwnd* of the epoch α is set to *cwnd*(0). W_{max} (called the current maximum or the origin point) is the *cwnd* where packet losses occurred previously. Refer to [17] for more details on C and K.



(b) Φ: Normal Distribution CDF

Figure 6: Adaptive selection of Φ based on CUBIC's *cwnd*.

at W_{max} previously. The CUBIC's heuristic indicates that the probability of packet loss is increasing fast at this stage. To counter the port blackout actively, **Strategy 2** is to reduce *rMin* aggressively.

Stage 3) Slow growth of *cwnd* (when *cwnd* $\geq W_{max}$): If the *cwnd* grows past W_{max}, CUBIC enters a max probing phase [17]. At the beginning of the max probing phase, the *cwnd* grows slowly to find out a new maximum point nearby as the CUBIC's heuristic expects that the probability of packet loss becomes higher when *cwnd* $\geq W_{max}$. Thus, as **Strategy 3**, *rMin* must decrease aggressively again to prevent the port blackout.

Stage 4) Fast growth of *cwnd* (when *cwnd* $\geq W_{max}$): If no packet loss is detected for some period of time after stage 3, CUBIC performs a fast increase of *cwnd* since it guesses the new maximum is far away. Thus, **Strategy 4** is to not reduce *rMin* actively at this stage.

4.2. Adaptive Algorithm to Calculate rMin

We adopt the proposed strategies discussed in Section 4.1 and propose the TCPRand's adaptation method (Algorithm 1) to calculate *rMin* before sending a packet.

1) How to decide rMin?

rMin is calculated based on $\Phi(x, \mu, \sigma^2)$, which is the normal distribution CDF[1] shown in Figure 6(b). As the first parameter of Φ, x is a normalized distance between *cwnd* and α as shown at the line 3 of Algorithm 1. For instance, if *cwnd* = W_{max}, x = 1. The second and third parameters of Φ, (i.e., μ and σ^2) are the mean and the variance, respectively and they are configurable. *rMin* is determined by the line 5 of Algorithm 1 based on Φ and the other parameter θ, which is the lower bound of *rMin*. We set θ=200B to prevent too much goodput degradation and to keep reasonable fairness (see the tradeoff between fairness and goodput depending on *rMin* in Figure 8).

The normal distribution CDF supports our strategy for each of the 4 stages well as follows. Assume that $\mu = 1$. At stage 1, Φ increases very slowly and it leads to the gradual reduction of *rMin* as **Strategy 1**. At stage 2, Φ increases fast and finally converged to 0.5; it causes the fast reduction of *rMin* as **Strategy 2**. At stage 3, Φ grows quickly so that the reduction of *rMin* is still fast as **Strategy 3**. At stage 4, Φ grows leisurely and leads to the slow reduction of *rMin* as **Strategy 4**.

2) When to turn TCPRand on/off?

Trigger point: Based on **Strategy 1**, we activate TCPRand only when the $\tau \leq$ *cwnd*. The trigger point τ shown in Figure 6(a) is acquired by:

$$\tau = W_{max} - \frac{W_{max} - \alpha}{\nu_\tau} \tag{2}$$

where ν_τ is a scale factor tuning τ. If $\nu_\tau = 1$, $\tau = \alpha$. If $\nu_\tau \to \infty$, $\tau = W_{max}$.

End point: With **Strategy 4**, TCPRand can also set the end point ω, as shown in Figure 6(a).

[1] To reduce the Φ calculation overhead (not trivial) at kernel, we pre-calculated Φ for various input parameters and stored the result in a table. Thus, Φ is acquired by a simple table lookup.

Algorithm 1 Adaptation Method to Select *rMin*

1: **Input:** ω, τ, $cwnd$, μ, σ^2, θ
2: **if** $\tau \leq cwnd \leq \omega$ **then**
3: $\quad x = \dfrac{cwnd - \alpha}{W_{max} - \alpha}$ /* normalized distance from α */
4: $\quad \Phi(x, \mu, \sigma^2) = \dfrac{1}{2}\left(1 + \frac{1}{\sqrt{\pi}} \int_{-\left(\frac{x-\mu}{\sigma\sqrt{2}}\right)}^{\frac{x-\mu}{\sigma\sqrt{2}}} \mathrm{e}^{-t^2}\, \mathrm{d}t\right)$
5: $\quad rMin = \max(\mathrm{MSS} \times (1 - \Phi(x, \mu, \sigma^2)), \theta)$
6: **else**
7: $\quad rMin = \mathrm{MSS}$
8: **end if**

TCPRand is deactivated if *cwnd* grows above ω, which is set by:

$$\omega = W_{max} + \frac{W_{max} - \alpha}{\nu_\omega} \tag{3}$$

where ν_ω is a scale factor tuning ω. If $\nu_\omega \to 0$, $\omega \to \infty$. If $\nu_\omega = 1$, $\omega = 2 \times W_{max} - \alpha$. Preventing ω from growing too much is useful to avoid unnecessary payload size randomization in case of large *cwnd* (e.g., when the competing flows finish). Note that Eqs. (2) and (3) are implemented at line 2 in Algorithm 1.

5. Evaluation Setup

We evaluate the proposed solution in two ways: ns-3 simulator and real testbed. We first describe our evaluation environments, enumerate parameters for TCPRand, and finally outline evaluation metrics before presenting our results in Section 6 and 7.

5.1. NS-3 Simulation Environment

We incorporate TCPRand with the packet-level simulator ns-3 to experiment it in a full-blown topology (i.e., fat-tree [14]) of a data center network as shown in Figure 1. We choose ns-3 because it enables high performance simulation. We adopt most of the configuration parameters suggested in [2] (including link capacity (=1Gbps), TCP minRTO value (=2ms), MSS value (=1460B), routing policy, etc.). The processing delay of each switch is set to 25 microseconds as suggested in [18]. We integrate TCPRand to both NewReno and CUBIC whose source is available at [19]. In the remainder of the paper, CUBIC and TCP are interchangeably used unless otherwise mentioned.

5.2. Testbed Environment

To make our testbed realistically reflect a fat-tree topology (shown in Figure 1), we use a topology illustrated in Figure 7. All the machines, on which TCPRand is running, are equipped with an Intel Core i7-3770K CPU @3.50GHz, 32GB of main memory and Intel 82579 Gigabit Ethernet NIC. We use two different types of switches: Cisco catalyst 2970 which adopts the simple drop-tail queue management policy and HP 5900 which supports ECN capability and enables us to run DCTCP for comparison with TCPRand. We implement TCPRand by modifying the TCP output engine in the Linux kernel 3.2.39. All the offload options including TCP segmentation offload (TSO), generic segmentation offload (GSO) and generic receive offload (GRO) are disabled because they use the offload engine in NIC and make TCPRand not work as expected. We evaluate the impact of disabling the options in Section 7.

TCPRand randomizes the payload size, which in most cases becomes smaller than MSS, and as a result it may generate more packets compared to the regular TCP. Due to its unique characteristics, we consider the following factors that can affect the performance of TCPRand as follows:

Appropriate Byte Counting (ABC): Even though TCP output engine in Linux increases *cwnd* based on the number of acks (which works well with the MSS-sized payload), by enabling ABC [20] option, *cwnd* increases based on the "bytes" acked. In Linux kernels, ABC is implemented only in Reno but we also implement it in CUBIC to observe its effects. However, for the scenarios where TCP outcast happens (e.g., many flows and a few flows are arriving at two input ports and destined to the same output port), the use of ABC did not change the overall test result. It is because the effect of ABC is far smaller than that of the port blackout in the TCP outcast scenarios. Thus, in this paper, we only show the results experimented without ABC.

Nagle's algorithm and congestion control: To observe how TCPRand cooperates with different congestion control mechanisms, we choose Reno, BIC and CUBIC [17] and test them with or without the Nagle's algorithm [21]. However, for the TCP outcast scenarios, there is no noticeable difference among the six combinations since the port blackout overwhelms their effect. Thus, we only address the case with CUBIC and the Nagle's algorithm since CUBIC is the default congestion management pro-

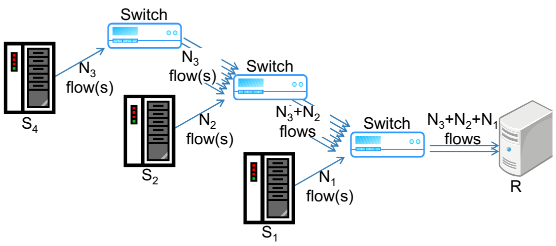


Figure 7: Abstracted subset topology of fat-tree in Figure 1.

tocol in Linux today and most bulk transfer applications enable the Nagle's algorithm.

SACK: By default, SACK is enabled for the fast recovery from multiple packet losses in today's Linux. However we also conduct experiments without SACK to see its role in TCP outcast scenarios when combined with TCPRand.

5.3. TCPRand Parameters

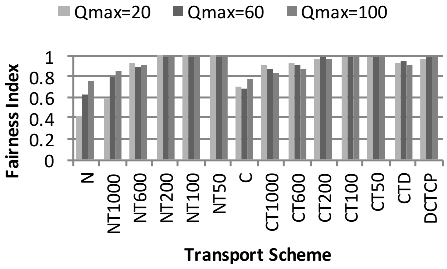
TCPRand has four parameters (i.e., σ^2, μ, ν_τ, ν_ω), thus allowing many possible combinations of these parameters. For instance, we can vary parameter values as follows: σ^2={0.2, 1, 5}, μ={1, 0}, ν_τ={∞, 1}, ν_ω={0, 1}. A larger σ^2 causes faster growth of Φ when $x < \mu$ but Φ grows slowly when $x \geq \mu$. With a smaller μ, more aggressive increase of Φ can be observed. $\tau = \alpha$ if $\nu_\tau = 1$, while $\tau = W_{max}$ if $\nu_\tau \to \infty$. $\omega = 2 \times W_{max} - \alpha$ if $\nu_\omega = 1$, while $\omega \to \infty$ if $\nu_\omega = 0$. Out of many configurations possible, we conduct evaluation with the three sets of configurations denoted in the form of (σ^2, μ, ν_τ, ν_ω). One configured as (1, 1, 1, 1) represents a moderate setting, which is our default setting. The other is set as (1, 1, ∞, 1) which represents the most conservative setting. The third is the most aggressive setting that is configured as (1, 0, 1, 0). Unless otherwise mentioned, we use the default setting while we mix and match the configurations when necessary.

5.4. Evaluation Metrics

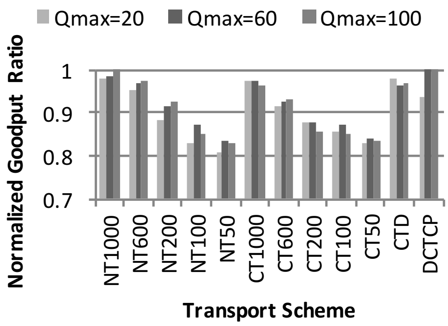
We are primarily interested in evaluating TCPRand with two key metrics: fairness and goodput across both real testbed and simulation cases. We shortly define each of them next.

Fairness: We use Jain's fairness index [22] defined as follows:

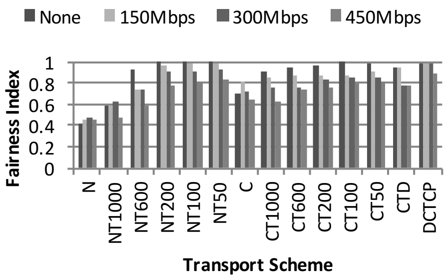
$$Fairness(g_1, g_2, \cdots, g_n) = \frac{\left(\sum_{i=1}^{n} g_i\right)^2}{n \times \sum_{i=1}^{n} g_i^2} \qquad (4)$$



(a) Fairness by different Q_{max}



(b) Total goodput by different Q_{max}



(c) Fairness by different background traffic amount

Figure 8: Effect of Q_{max} and background traffic. N: NewReno, NTx: NewReno+TCPRand(*rMin*=x bytes), C: CUBIC, CTx: CUBIC+TCPRand(*rMin*=x bytes), CTD: CUBIC+ Adaptive TCPRand with (σ^2, μ, ν_τ, ν_ω)=(1, 1, 1, 1) and DCTCP.

where g_i is the average goodput of flows sent by S_i. Note that in the ideal case, fairness index is 1.

Goodput: As typically defined, we obtain goodput by dividing the amount of application-level data by the total time taken until the completion of its delivery. Total goodput is the sum of all flows' goodputs.

In addition to the two key metrics discussed above, we also show other interesting metrics such as flow completion time (FCT) (which is especially important for short flows), consecutive packet losses, timeouts (in terms of frequency and period) and flow convergence trend (to show the stability of TCPRand flows compared to TCP ones).

6. NS-3 Simulation Results

We evaluate TCPRand in an ns-3 environment. First, the evaluation focus lies on the two metrics (i.e., fairness and goodput) while we vary network conditions such as switch queue size (Q_{max}) and the amount of background traffic. Second, we measure the timeouts, the main cause of TCP outcast, in two different aspects: frequency and period. Third, we show how tolerable TCPRand is to TCP outcast varying the number of competing flows. Fourth, we conduct simulation with data center workloads [23] to show that TCPRand in general supports flows with different sizes well.

6.1. Fairness and Goodput Analysis

A total of 15 senders (S_1-S_{15}) generate one TCP flow per sender to receiver R in the fat-tree topology in Figure 1. We check how TCPRand mitigates the TCP outcast problem. In doing so, we analyze how TCPRand interacts with varying the maximum length (Q_{max}, expressed as the number of packets) of the drop-tail queue and background traffic values. Specifically, each sender simultaneously generates a flow for 10 seconds. Each flow sent from sender S_n is denoted by F_n. Thus, in the fat-tree, E_1 is the most bottlenecked switch and F_1 is the most outcast flow since F_1 competes with $F_{2:15}$ for the output queue at E_1.

We additionally plot the results of TCPRand with static settings (i.e., fixed *rMin*) alongside TCPRand (denoted as CTD in Figure 8) to demonstrate why the adaptive *rMin* selection method is better than configuring *rMin* statically. We also compare TCPRand with DCTCP, a representative cross-layer protocol for data center networks whose congestion control mechanism requires additional switch support including random early marking and Explicit Congestion Notification (ECN). *We do this comparison in order to shed light on how close the performance of a pure transport layer solution like TCPRand can be to that of a cross-layer approach like DCTCP.* For DCTCP, we set a marking threshold to $0.2 \times Q_{max}$ as proposed and used for 1Gbps link in [9].

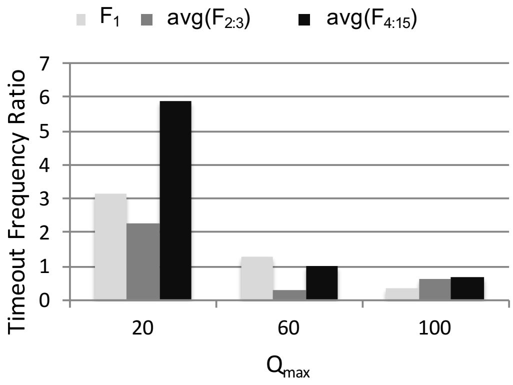
6.1.1. Impact of Q_{max} on fairness and goodput

To see the effect of Q_{max} to TCPRand, we set $Q_{max} = \{20, 60, 100\}$ in the unit of packet. Notations for transport schemes are given in the caption of Figure 8. As shown in Figure 8(a), the regular TCP (i.e., N and C) suffers from the unfairness caused by the TCP outcast. As decreasing *rMin* statically, the outcast flows recover quickly and the fairness index approaches to 1 regardless of Q_{max}. However, more aggressive reduction of *rMin* triggers more loss of total goodput as shown in Figure 8(b) (goodput ratio normalized to that of N or C). For instance, given $Q_{max} = 100$, as *rMin* decreases, the fairness of CTx increases (0.976, 0.991 and 0.996 with CT200, CT100 and CT50, respectively). However, there are consistent decreases in goodput of CTx: (0.855, 0.85 and 0.835 with CT200, CT100 and CT50, respectively).

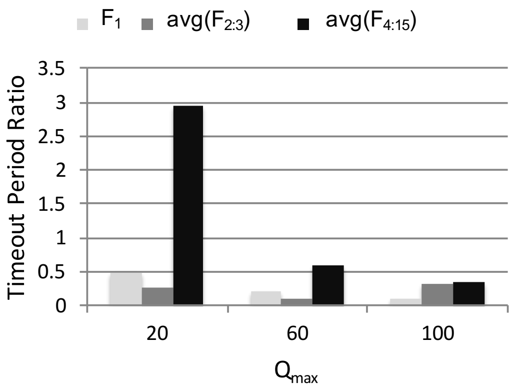
Overall, DCTCP shows good balance between fairness and goodput. DCTCP can minimize packet drop itself by keeping the queue length short with the help of switch's ECN capability, whereas TCPRand promotes *fair packet drops* among flows through the payload size randomization. Little difference in total goodput between CUBIC and DCTCP (see DCTCP bars at $Q_{max} = 60$ or 100 in Figure 8(b)) is because CUBIC flows fully utilize the link capacity at an aggregated level and so do DCTCP flows. However, when $Q_{max} = 20$, DCTCP loses the goodput considerably (i.e., 0.936). It is because the marking threshold (i.e., $4 = 0.2 \times 20$) is too small for senders to increase its *cwnd* high enough to acquire full goodput by the DCTCP congestion control algorithm.

6.1.2. Impact of background traffic on fairness

For this simulation, given 15 senders, we make each sender additionally generate, to the receiver, 10, 20 and 30Mbps UDP CBR traffic, accounting for 150, 300 and 450Mbps aggregate background traffic, respectively. Figure 8(c) shows the effect of background traffic to the fairness where $Q_{max} = 20$. We clearly observe that TCPRand always achieves higher fairness than the regular TCP. However, the larger the background flows, the smaller the additional fairness gain of TCPRand to the regular TCP. Note that in this simulation, the payload size of the background flows is not randomized.



(a) Timeout frequency ratio of TCPRand to TCP



(b) Timeout period ratio of TCPRand to TCP

Figure 9: The effect of TCPRand to timeouts. F_1, avg($F_{2:3}$) and avg($F_{4:15}$) in legend indicate the timeout statistics from the 2-hop flow (the most outcast flow), an average of 4-hop flows (F_2 and F_3) and an average of 6-hop flows (from F_4 to F_{15}) in Figure 1, respectively.

Thus the effect of the payload size randomization to the port blackout is restricted more as the amount of background traffic increases. However, even with the largest background traffic (i.e., 450Mbps), TCPRand still achieves a noticeable fairness improvement. DCTCP also achieves high fairness (i.e., 0.887) under the same background traffic condition.

6.2. Timeout

To investigate the exact reason of the fairness increase caused by TCPRand, we measure two metrics regarding timeout: frequency and period. Note that the former indicates the number of timeouts triggered while the latter does the total amount of time halted by the timeouts. Using the results from the simulation performed in Section 6.1, we compare the timeout frequency and timeout period of TCPRand (i.e., CTD) to the regular TCP (i.e., CUBIC) in Figure 9.

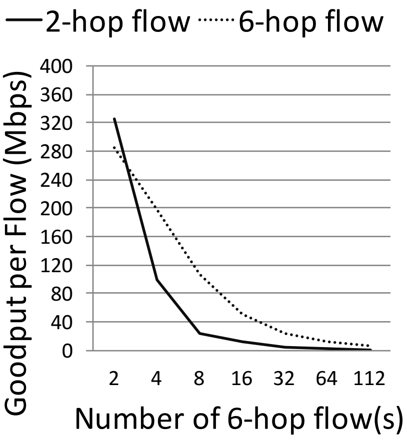
Timeout frequency: Since TCPRand generates (smaller but) more packets than TCP, it causes more packet losses than TCP. As shown in Figure 9(a), when the Q_{max} is small (i.e., 20), this property makes all TCPRand flows (regardless of the senders' locations on the topology in Figure 1) experience more timeouts than TCP ones. However, as Q_{max} increases, TCPRand reduces the timeout frequency more than TCP. For instance when $Q_{max} = 100$, all TCPRand flows experience even less timeouts than TCP ones. It turns out that the shuffle effects triggered by TCPRand at bottleneck queue further decreases consecutive packet losses (the main cause of timeout) at each input port.

Timeout period: More importantly, compared to TCP, TCPRand always reduces the timeout period of the outcast flow (F_1) regardless of Q_{max} (see Figure 9(b)). This result may look contradictory to what is shown in Figure 9(a). For instance, when $Q_{max} = 20$, TCPRand increases the timeout frequency of F_1 by a factor of $\sim$3 but decreases the timeout period of F_1 by half compared to TCP. This is because *TCPRand reduces the consecutive timeouts, and hence keeps RTO small.* In other words, consecutive timeouts trigger the exponential backoff to the retransmit timer; preventing them makes it possible to drastically decrease the timeout period. TCPRand is effective in preventing such consecutive timeouts for the outcast flow, thus decreasing its overall timeout period. Furthermore, as Q_{max} increases (i.e., $Q_{max} \geq 60$), TCPRand reduces the timeout period of all the flows even including the non-outcast ones (F_4-F_{15}).

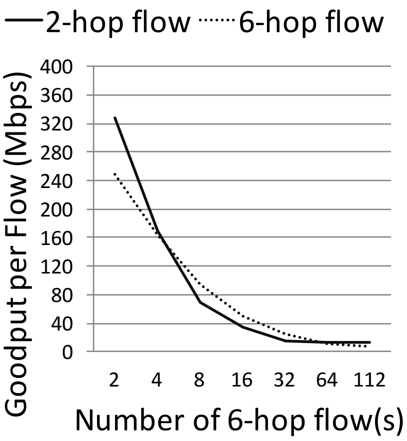
6.3. Influence of Different Number of Senders

To understand how TCPRand operates at the bottleneck queue under a varying number of senders, we use a larger fat-tree that comprises of 8-port switches. A 8-ary fat-tree topology consists of 80 switches and 128 hosts. We make a 2-hop flow compete with the different numbers of 6-hop flows[2]. Note that given a receiver, there are all 112 6-hop

[2]For simplicity, we do not generate 4-hop flows for this simulation.



(a) CUBIC

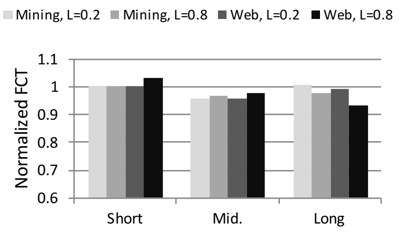


(b) TCPRand

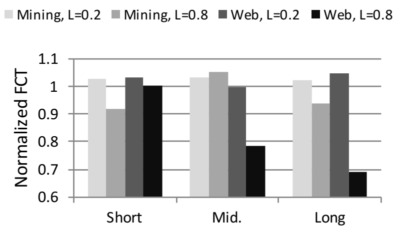
Figure 10: Goodput of CUBIC and TCPRand flows when a 2-hop flow competes with different number of 6-hop flow(s) in 8-ary fat-tree topology where there are 128 servers.

senders in the topology. In the simulation, Q_{max} is set to 100.

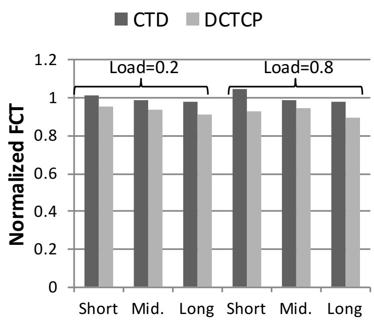
Figure 10 shows how TCPRand affects the goodput of a 2-hop and 6-hop flow(s). From the figure, we first observe that the 2-hop flow acquires more goodput than the 6-hop flow(s) when the number of the 6-hop flow(s) is ≤ 2. This is because in general TCP throughput is proportional to the inverse of RTT [24]. However, as the number of the 6-hop flows increases, the 2-hop CUBIC flow starts to suffer from TCP outcast as shown in Figure 10(a). On the other hand, the goodputs of TCPRand flows agree to their fair share of bandwidth well regardless of the number of 6-hop flows, and the TCP outcast problem is successfully mitigated even in a larger topology (see Figure 10(b)).



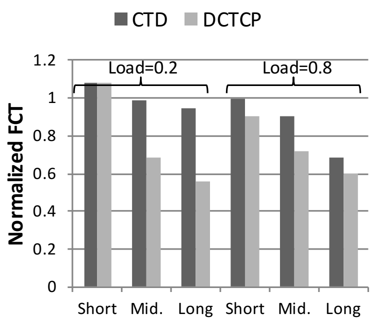
(a) Average FCT for data mining workload



(b) 100 percentile FCT for data mining workload



(c) Average FCT for web search workload



(d) 100 percentile FCT for web search workload

Figure 11: FCT of TCPRand and DCTCP normalized to that of CUBIC for different workloads and traffic loads per flow size group. Sizes of short, mid and long flows are [0, 100KB), [100KB, 10MB), and [10MB, ∞), respectively.

6.4. Analysis with Real Data Center Workloads

Since TCPRand tends to keep the payload size less than MSS, it may increase flow completion time (FCT), in particular that of short flows, which in general originates from latency-sensitive applications. To answer that question, we trace the effect of TCPRand to FCT using two realistic data center workloads (i.e., web search and data mining) [23] that consist of a mix of short and long flows. Flow arrivals follow a Poisson process, and the sender and receiver for each flow are chosen randomly among all the 16 end-nodes (i.e., R and S_1-S_{15} in Figure 1). The flow arrival rate (i.e., load in the fabric) is varied from 0.2 to 0.8 as suggested in [23].

Figure 11 shows the FCT of TCPRand and DCTCP normalized to CUBIC per flow size group. Regarding TCPRand, two trends are observed while DCTCP in general is efficient in decreasing FCT. First, TCPRand increases neither the average nor the 100 percentile FCT of short flows in both web search and data mining workloads noticeably. It is because many short flows are extremely small in real (especially in data mining) workloads and many of them finish before TCPRand performs the aggressive reduction of *rMin*. Second, under high traffic load (i.e., Load=0.8), the CUBIC (especially long) flows often suffer from timeouts, whereas TCPRand is in general effective in suppressing timeouts. Hence, long TCPRand flows in general achieve shorter FCTs than CUBIC flows under high traffic load. Moreover, since the web search workload contains more long flows than the data mining one and TCPRand reduces the timeout period of that long flows, the 100 percentile FCT (especially of long flows) of TCPRand under high traffic load decreases as shown in Figure 11(d).

7. Experimental Results

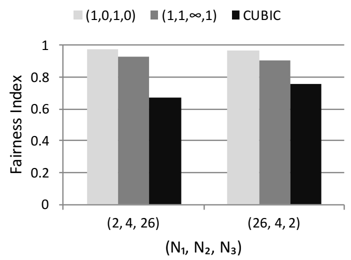
We now evaluate TCPRand in a real testbed. The main purpose of this evaluation in the testbed is to confirm that TCPRand in practice improves fairness without compromising goodput in the presence of TCP outcast. Next, we conduct microscopic analysis to shed light on how several aspects (packet drops, timeouts, and retransmissions) in TCP congestion control are affected by TCPRand.

We construct a testbed which simplifies the fat-tree topology in Figure 1 but still preserves its essential nature for creating TCP outcast. The testbed topology is shown in Figure 7. This topology allows us to create many TCP outcast cases with different combinations of (N_1, N_2, N_3) where N_1, N_2 and N_3 are the number of flows generated by S_1, S_2 and S_4, respectively in Figure 7. In fact, we tested TCPRand in many TCP outcast events and found in all cases TCPRand achieves similar fairness and goodput. Thus, out of them, we choose two combinations: *i)* (2, 4, 26) for mimicking the observation in [2] that more flows come from distant senders while less flows come from close senders in the fat-tree and *ii)* (26, 4, 2) as the opposite of case i) to show that TCPRand can solve the TCP outcast problem even in unusual situations.

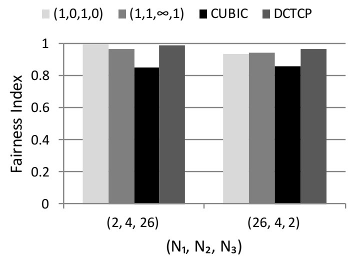
7.1. Fairness and Goodput Analysis

Fairness: Figures 12(a) and 12(b) show that regardless of (σ^2, μ, ν_τ, ν_ω) configurations, TCPRand always achieves a higher fairness index than CUBIC. While not shown for brevity, we measure the fairness with many other combinations of the configuration parameters and observe that higher fairness is in general achieved as configurations become more aggressive (i.e., with smaller μ and τ, or larger ω) in randomizing the payload. Even with the most conservative setting (1, 1, ∞, 1), TCPRand still obtains 16-41% higher fairness than CUBIC. Moreover, regardless of the configuration parameters or switch types, TCPRand always guarantees 0.9 or higher fairness index in all the scenarios we experimented. In addition, we also observe that TCPRand's fairness index is comparable to DCTCP's in Figure 12(b).

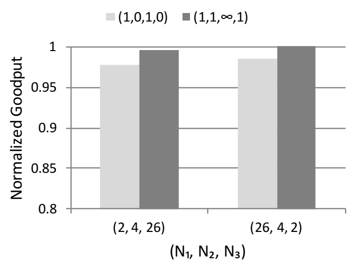
Loss of total goodput: If $\mu = 1$, TCPRand always keeps the additional loss of total goodput to CUBIC low (<1% in Figure 12(c) and <4% in Figure 12(d)). Although we do not show the exact picture for brevity, even for the case where TCP outcast does not happen (i.e., the same number of flows compete) and the total number of competing flow is small (i.e., 3), TCPRand minimizes the total goodput loss ($\sim$1%) effectively. This indicates that even though TCPRand is mainly designed to pursue more fairness for TCP outcast scenarios, it causes only a trivial amount of additional total goodput loss for non-outcast scenarios; this is possible since the proposed adaptive randomization scheme in Algorithm 1 avoids unnecessary payload size randomization. In the worst case, compared to CUBIC, the additional loss of total goodput is marginal ($\sim$2.3% in Figure 12(c) and $\sim$4.7% in Figure 12(d)). This level of goodput loss can be acceptable as well because most many-to-one applications that are barrier synchronized [12] may



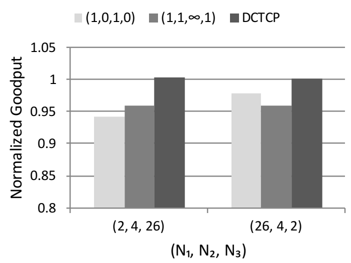
(a) Fairness index, Cisco Catalyst 2970



(b) Fairness index, HP 5900



(c) Normalized goodput, Cisco Catalyst 2970



(d) Normalized goodput, HP 5900

Figure 12: Fairness and total goodput of TCPRand, CUBIC and DCTCP under the testbed with the topology in Figure 7, respectively. The 4-tuple in legend corresponds to (σ^2, μ, ν_τ, ν_ω) of TCPRand. (1, 0, 1, 0) is the most aggressive setting while (1, 1, ∞, 1) is the most conservative configuration. We use two different types of switches (Cisco Catalyst 2970 and HP 5900) to confirm that TCPRand is effective to TCP outcast in various hardware settings. Note that HP 5900 is ECN-capable and thus allows DCTCP experiments.

improve their job completion time performance by enhancing the goodput of the slowest TCP connection rather than maximizing total goodput. The total goodput of DCTCP is similar to that of CUBIC under the outcast scenarios (see Figure 12(d)).

7.2. Microscopic Analysis on Improved Fairness

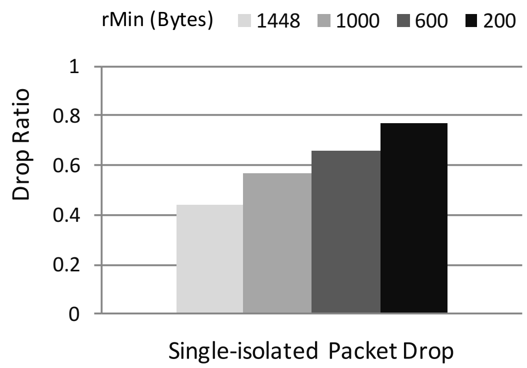
To further understand what effects TCPRand brings to TCP flows in detail, we conduct a microscopic analysis with a simplest topology exhibiting the TCP outcast problem. We do this in our testbed instead of ns-3 simulator since the testbed environment can best reflect the microscopic behaviors caused by the temporal port blackout that happens at an output queue of commodity hardware switches.

We use the same testbed shown in Figure 7 where we only use two senders (S_1 and S_2) and one receiver (R). S_1 creates one flow (denoted as F_1) to R and S_2 does M flows (from F_2 to F_{M+1}, denoted as $F_{2:M+1}$) to the same R. We vary M where M={5, 10, 15, 20, 25}. Out of these five cases, we only present the most prominent results that are observed when M={5, 15, 25}. We disable the adaptive *rMin* selection method and statically set *rMin* values. *rMin* of each flow is set to 1,448, 1,000, 600 or 200 bytes to make our analysis more tractable. For the measurements, we use *iperf* and run it for 100 seconds per each case. All flows (i.e., $F_{1:M+1}$) start transmission simultaneously[3]

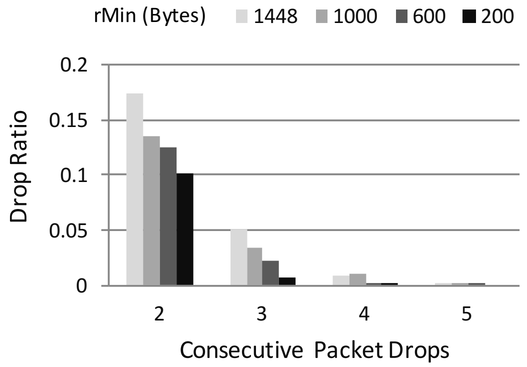
Basically, SACK is enabled in our experiments as most modern Linux distributions support SACK by default, but for a broader analysis, we also present results while disabling SACK as well. We examine consecutive packet drops, TCP timeouts, and packet retransmission for the analysis.

Consecutive packet drops: Figure 13 shows the distribution of packet drops that the outcast flow experiences with SACK option when M=15. For the experiment, both S_1 and S_2 use the same *rMin*. As shown in Figure 13(a), as *rMin* decreases, the ratio of *single-isolated* packet drops to the total packet drops increases. Omitted for brevity, the largest increase is observed with the smallest *rMin* (i.e., 200B) across all M's. This indicates that more aggressive payload size randomization prevents consecutive packet drops more effectively. Figure 13(b) shows the detailed distribution of consecutive (i.e.,

[3]Note that we also conducted experiments by varying the arrival times of some flows and found no visible difference in the results.

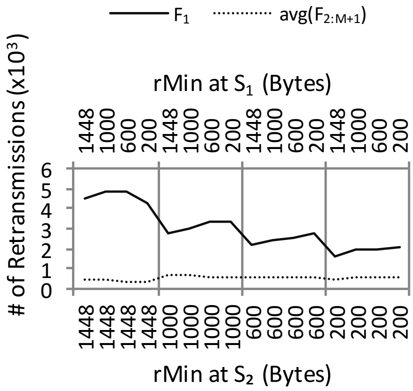


(a) Ratio of single-isolated packet drops to total packet drops

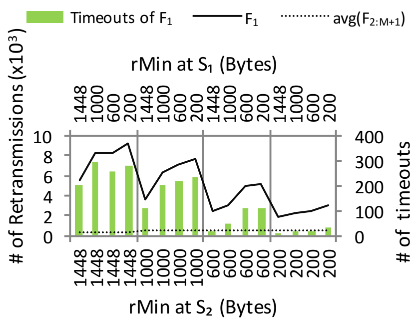


(b) Ratio of consecutive packet drops to total packet drops

Figure 13: The outcast flow's packet drop distribution with SACK when M=15 using Cisco Catalyst 2970.



(a) With SACK



(b) Without SACK

Figure 14: The number of TCP timeouts and retransmissions when M=15 (with Cisco Catalyst 2970). Note that no TCP timeout is observed with SACK.

$2 \sim 5$) packet drops. It is clearly observed that the frequency of consecutive packet drops decreases dramatically (especially for more than two consecutive drops) as *rMin* decreases. When SACK is off, the number of consecutive packet drops decreases considerably up to M=15, and stops decreasing as M further increases.

TCP timeouts: Figure 14 shows that TCPRand+SACK prevents the outcast flow from experiencing any TCP timeout; although omitted for brevity, only one configuration caused at most 4 timeouts when M=25. On the other hand, disabling SACK shows two intriguing patterns in Figure 14(b). *i)* TCPRand reduces the number of TCP timeouts enormously with smaller *rMin* values; when M=15, TCP timeouts decrease from 204 (*rMin*=1,448, regular TCP) to 9 times. *ii)* However, when M grows to 25, TCPRand fails to reduce TCP timeouts noticeably. Even for the regular TCP, enabling SACK option greatly helps reduce the number of TCP timeouts (e.g., only one timeout when M=25). This is because SACK makes a flow recover efficiently against multiple packet losses. However, there still exists unfairness (as illustrated with Figure 12(b)) since the outcast flow must recover from multiple packet losses alone while the non-outcast flows share the recovery burden among themselves.

Packet retransmissions: Figure 14 shows the number of packet retransmissions of flows (represented by the left y-axis in each graph). We make the following observations:

First, when *rMin* of the non-outcast flows (i.e., $F_{2:16}$ at S_2) is fixed, decreasing *rMin* of the outcast flow (i.e., F_1 at S_1) increases the number of packets retransmitted by the outcast flow. For instance, see in Figures 14(a) and 14(b) a configuration where *rMin* of $F_{2:16}$ is set to 1,000B: as *rMin* of F_1 reduces from 1,448B to 200B, F_1 has an increasing number of retransmissions. This is because decreasing *rMin* usually makes TCPRand generate more packets (smaller than MSS) than the regular TCP.

Second, when *rMin* of the outcast flow (F_1) is fixed, reducing *rMin* of the non-outcast flows ($F_{2:16}$) tends to decrease the number of packet retransmissions in F_1. For example, from the solid line in Figure 14(a), compare points where (*rMin* of F_1, *rMin* of $F_{2:16}$) are (1,000, 1,448), (1,000, 1,000), (1,000, 600) and (1,000, 200); we clearly observe a decreasing trend in the number of retransmissions of F_1.

Third, Figures 14(a) and 14(b) show that the lack of selective acknowledgement mechanism makes TCPRand of the outcast flow retransmit more packets. In contrast, when M < 15 (the graph is omitted), we observe that TCPRand without SACK has the similar pattern to that with SACK.

The final observation is that in all cases, the outcast flow (F_1 at S_1) does more packet retransmissions than the non-outcast flows ($F_{2:16}$ at S_2) as expected.

Throughout the analysis, we find out that TCPRand in general decreases the number of consecutive drops, TCP timeouts and packet retransmissions of the outcast flow. Another interesting finding is that TCPRand alongside SACK option is most effective in alleviating several adversary events to TCP performance. However, even with SACK, statically changing *rMin* value is insufficient to completely address the TCP outcast problem, reassuring that our adaptive payload size randomization method is absolutely necessary.

8. Further Considerations

8.1. TCP Incast

TCP incast is another important problem which shares high similarity with TCP outcast problem. Hence, it is a natural question to ask whether or not TCPRand adversely affects the TCP incast problem. To answer that question, we perform a simulation to compare the goodput of CUBIC, TCPRand and DCTCP under a TCP incast scenario. In the simulation, the file size (or block size) is 128 KB and $Q_{max} = 100$.

Figure 15 shows that DCTCP in general achieves better goodput gain than the other two (CUBIC and TCPRand). This is mainly because DCTCP keeps queue length small (thus, mitigating packet drop itself) whereas TCPRand only promotes fair

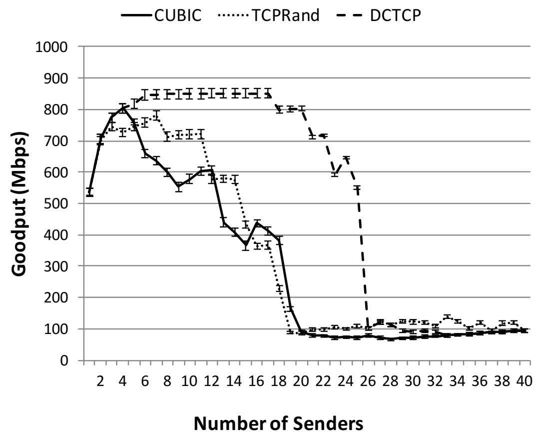


Figure 15: Comparison of TCP, TCPRand and DCTCP in a TCP incast scenario ($Q_{max} = 100$ and block size = 128KB). We use a simple star topology composed of 40 senders, one receiver and a 1GbE switch among them.

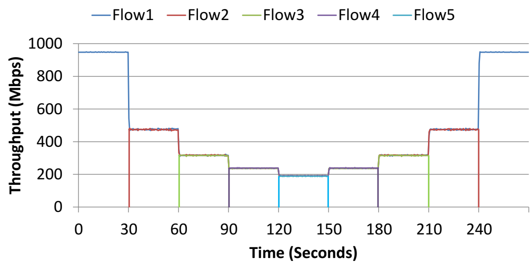
packet drops. In comparison with CUBIC, we find that TCPRand does not make the TCP incast problem worse than CUBIC. We also observe the similar trends (the graph is omitted) with different parameter values such as Q_{max} and block sizes.

8.2. Flow Convergence

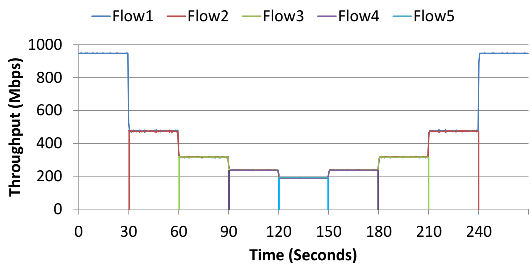
Since fair sharing of network bandwidth is one of the key characteristics of TCP, it is important to check if the payload size randomization process of TCPRand violates this property. Thus, we experiment how TCPRand flows converge on a simple testbed composed of five senders (from each of which one flow is generated), one receiver and a 1GbE switch among them. Each flow starts sequentially with 30 second interval and have different durations (i.e., 270s, 210s, 150s, 90s, and 30s), respectively as done in [25]. Figure 16(a) and 16(b) show how TCP and TCPRand flows converge, respectively. As shown in the figure, the convergence trends of TCP and TCPRand flows are quite similar, assuring that flow convergence in TCPRand is comparable to that of TCP.

8.3. Congestion Window Variation

To observe the effect of payload size randomization to the congestion window, we compare the congestion window variation of CUBIC and TCPRand in our testbed and show the result in Figure 17. Each flow (i.e., Flow1 and Flow2) in Figure 17 is generated from different senders at the same



(a) Convergence of TCP flows



(b) Convergence of TCPRand flows

Figure 16: Flow convergence in terms of network bandwidth sharing.

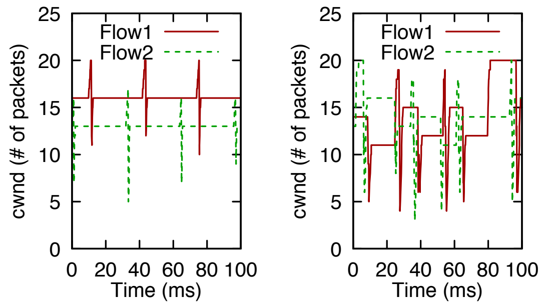
	No. of concurrent flows			
	1	10	100	1000
CUBIC	4.6%	5.3%	7.9%	10.2%
TCPRand	12.5%	16.1%	28.0%	42.2%

Table 1: CPU overheads of CUBIC and TCPRand.

time and destined to one receiver. As shown in Figure 17(a), the congestion window variation of the two CUBIC flows is rather synchronized, explaining why TCP is prone to port blackout, the main cause of TCP outcast. However, the congestion window of TCPRand shown in Figure 17(b) varies more asynchronously, which evidently shows the reduction of port blackout probability.

8.4. CPU Overhead

By default in Linux, offload options such as TSO are enabled to reduce CPU overhead if its NICs support them. On the other hand, TCPRand requires to switch off those options, and thus can require more CPU cycles. Here we measure the amount of CPU resources used by TCPRand and compare it with that of CUBIC measured while TSO is enabled. In our testbed, one CUBIC flow consumes 4.6% of the resources of one core, whereas one



(a) CUBIC (b) TCPRand

Figure 17: Congestion window variation.

TCPRand flow consumes 12.5%. As the number of concurrent flows increases, the CPU overheads in both schemes grow linearly (see Table 1). For instance, 100 flows make CUBIC and TCPRand consume 7.9% and 28.0% of the resources of one core, respectively. In an extreme case where there exist 1,000 flows in a host, the CUBIC flows consume 10.2% and the TCPRand flows consume 42.2% of the resources of one core. While TCPRand consumes more CPU resources than CUBIC, those amounts of CPU clock consumption may be acceptable since commodity servers are equipped with multicore CPUs.

9. Related Work

Link layer solutions: Random early detection (RED) [26] and stochastic fair queueing (SFQ) [8] have been tested to solve the TCP outcast problem. Prakash et al. [2] point out that RED shows RTT bias while SFQ makes flows have throughput fairly and achieves RTT fairness but uncommon in commodity switches. More importantly, a large-scale deployment of commodity off-the-shelf (COTS) switches enables low cost construction of data center networks. Unfortunately, these switches employ neither RED nor SFQ [2]. It would be prohibitively costly to replace them with high-end switches that are capable of exploiting these active queue management strategies. Zhang et al. [7] propose a protocol that supports bandwidth sharing by allocating switch buffer; the switch determines the size of the congestion window of its passing flow. However, all the switches in data centers must be modified for supporting such a feature to make use of this solution.

Network layer solutions: Equal-length routing [2] makes all flows from senders routed up to the core switch regardless of the senders' locations. Then, all the flows take the same downward path from the core to the destination which leads to RTT fairness. It uses a detour path to increase the path similarity instead of the shortest path. However, this approach causes performance degradation if data center networks are oversubscribed. Furthermore, it significantly lacks flexibility.

Transport layer solutions: The rate-based delivery (e.g., TCP pacing [10] and sending time randomization [12]) has also been considered as a solution to the TCP outcast problem. TCP pacing, combined with the window based congestion control, avoids burst delivery by giving some interval between the transmission times of two consecutive packets and shows inverse RTT bias. However, the TCP outcast problem still remains considerably in TCP pacing [2]. Chandrayana et al. propose a scheme randomizing the sending times by adjusting the inter-packet gap [11]. This, however, cannot retain the initial randomness created by the sender throughout the routing path mainly due to the bursty departure process at the first bottleneck queue. This makes the approach ineffective in a multi-hop environment. Moreover, the rate-based delivery has a severe practical limitation because it is practically infeasible to do (sub-)microsecond level packet spacing [27] (e.g., in 1/10Gbps link), quite strictly required to get better randomness effects in data center networks (where RTT $<$ 1ms [5]). Even though a high resolution timer (e.g., hrtimer in Linux) is available, operating systems hardly guarantee the precise control of inter-packet spacing time. Furthermore, frequent timer interrupts lead to a large interrupt handling overhead [5].

Hybrid solution: Alizadeh et al. propose DCTCP [9], which is a cross-layer (i.e., link layer + transport layer) approach. In comparison with DCTCP , we observed that DCTCP is effective to mitigate the TCP outcast problem by controlling a congested port's queue length properly. However, DCTCP must leverage random early marking and Explicit Congestion Notification (ECN) capability, which are not yet widely supported by most commodity ToR switches especially in small and medium data centers to our knowledge.

Per-packet scheduling: There have been several recent proposals on per-packet scheduling [23, 28, 29, 30]. These new datacenter transports are known to achieve near-optimal flow completion times. Hence, they are likely to mitigate effectively pathological congestion collapses such as TCP incast and outcast. However, a forklift upgrade is inevitable, meaning that all of the datacenter components (hosts and switches) should unanimously support one such scheme. Whereas this constraint may not be an issue in private datacenters, it can be a challenging problem in public cloud datacenters (e.g., Amazon EC2) where tenants can run different kinds of transport protocols in their virtual machines. On the other hand, pure transport approaches like TCPRand can be incrementally deployed (e.g., application by application) for cloud datacenter environments.

10. Conclusion

We proposed a payload size randomization scheme called TCPRand to address the TCP outcast problem in data center networks. TCPRand is a pure transport layer solution, which is easily-deployable and practical to the TCP outcast problem. Without relying on any special link layer support such as ECN, TCPRand guarantees superior enhancement of TCP fairness by reducing the timeout period of the outcast flow. Furthermore, it rarely sacrifices the total goodput since TCPRand avoids unnecessary payload size randomization. The flow convergence of TCPRand is also comparable to that of TCP. We envision that integrating TCPRand into TSO engine in NICs can reduce the CPU overhead, and leave it as future work.

References

[1] D. Nagle, D. Serenyi, A. Matthews, The Panasas ActiveScale Storage Cluster: Delivering Scalable High Bandwidth Storage, in: ACM/IEEE Supercomputing, 2004.

[2] P. Prakash, A. Dixit, Y. C. Hu, R. Kompella, The TCP Outcast Problem: Exposing Unfairness in Data Center Networks, in: USENIX NSDI, 2012.

[3] Y. Chen, R. Griffith, J. Liu, R. H. Katz, A. D.Joseph, Understanding TCP Incast Throughput Collapse in Datacenter Networks, in: WREN, 2009.

[4] A. Phanishayee, E. Krevat, V. Vasudevan, D. G. Andersen, G. R. Ganger, G. A. Gibson, S. Seshan, Measurement and Analysis of TCP Throughput Collapse in Cluster-based Storage Systems, in: USENIX FAST, 2008.

[5] V. Vasudevan, A. Phanishayee, H. Shah, E. Krevat, D. G. Andersen, G. R. Ganger, G. A. Gibson1, B. Mueller, Safe and Effective Fine-grained TCP Retransmissions for Datacenter Communication, in: ACM SIGCOMM, 2009.

[6] J. Lee, Y. Turner, M. Lee, L. Popa, S. Banerjee, J.-M. Kang, P. Sharma, Application-driven Bandwidth Guarantees in Datacenters, in: ACM SIGCOMM, 2014.
[7] J. Zhang, F. Ren, X. Yue, R. Shu, C. Lin, Sharing bandwidth by allocating switch buffer in data center networks, IEEE JSAC 32 (1) (2014) 39–51.
[8] P. E. McKenney, Stochastic fairness queueing, in: IEEE INFOCOM, 1990.
[9] M. Alizadeh, A. Greenberg, D. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, M. Sridharan, Data Center TCP (DCTCP), in: ACM SIGCOMM, 2010.
[10] A. Aggarwal, S. Savage, T. Anderson, Understanding the Performance of TCP Pacing, in: IEEE INFOCOM, 2000.
[11] K. Chandrayana, S. Ramakrishnan, B. Sikdar, S. Kalyanaraman, On randomizing the sending times in tcp and other window based algorithms, Computer Networks 50 (3) (2006) 422–447.
[12] H. Wu, Z. Feng, C. Guo, Y. Zhang, ICTCP: Incast Congestion Control for TCP in Data Center Networks, in: ACM CoNEXT, 2010.
[13] `http://www.nsnam.org`.
[14] M. Al-Fares, A. Loukissas, A. Vahdat, A scalable commodity datacenter network architecture, in: ACM SIGCOMM, 2008.
[15] IEEE Std 802.3-2002, IEEE Standard for Information technology— Telecommunications and information exchange between systems— Local and metropolitan area networks— Specific requirements, Part 3: Carrier sense multiple access with collision detection (CSMA/CD) access method and physical layer specifications (2002).
[16] R. Kapoor, A. Snoeren, G. Voelker, G. Porter, Bullet trains: a study of NIC burst behavior at microsecond timescales, in: ACM CoNEXT, 2013.
[17] I. Rhee, L. Xu, CUBIC: A new TCP-friendly high-speed TCP variant, in: PFLDNet Workshop, 2005.
[18] Design Best Practices for Latency Optimization, Financial Services Technical Decision Maker White Paper, `http://www.cisco.com/application/pdf/en/us/guest/netsol/ns407/c654/ccmigration_09186a008091d542.pdf`.
[19] `http://www.nsnam.org/wiki/Current_Development`.
[20] `http://tools.ietf.org/html/rfc3465`.
[21] J. Nagle, Congestion control in IP/TCP internetworks, RFC896, Internet Engineering Task Force (1984).
[22] R. Jain, A. Durresi, G. Babic, Throughput Fairness Index: An Explanation, ATM Forum/99-0045 (1999).
[23] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, S. Shenker, pFabric: Minimal Near-Optimal Datacenter Transport, in: ACM SIGCOMM, 2013.
[24] J. Padhye, V. Firoiu, D. Towsley, J. Kurose, Modeling TCP Throughput: A Simple Model and its Empirical Validation, in: ACM SIGCOMM, 1998.
[25] G. Judd, Attaining the Promise and Avoiding the Pitfalls of TCP in the Datacenter, in: USENIX NSDI, 2015.
[26] S. Floyd, V. Jacobson, Random early detection gateways for congestion avoidance, IEEE/ACM ToN 1 (4) (1993) 397–413.
[27] C. Lee, K. Jang, S. Moon, Reviving Delay-based TCP for Data Centers, (Poster) in *ACM SIGCOMM* (2012).
[28] W. Bai, L. Chen, K. Chen, D. Han, C. Tian, W. Sun, PIAS: Practical Information-Agnostic Flow Scheduling for Data Center Networks, in: ACM HotNets, 2014.
[29] M. Grosvenor, M. Schwarzkopf, I. Gog, R. Watson, A. Moore, S. Hand, J. Crowcroft, Queues Dont Matter When You Can JUMP Them!, in: USENIX NSDI, 2015.
[30] P. Gao, A. Narayan, G. Kumar, R. Agarwal, S. Ratnasamy, S. Shenker, Phost: Distributed Near-optimal Datacenter Transport over Commodity Network Fabric, in: ACM CoNEXT, 2015.